# Response Time Service Level Agreements for Cloud-hosted Web Applications

Hiranya Jayathilaka    Chandra Krintz    Rich Wolski

Computer Science Dept., Univ. of California, Santa Barbara

{hiranya,ckrintz,rich}@cs.ucsb.edu

## Abstract

Cloud computing is a successful model for hosting web-facing applications that are accessed by their users as services. While clouds currently offer Service Level Agreements (SLAs) containing guarantees of availability, they do not make performance guarantees for deployed applications.

In this work we present Cerebro – a system for establishing statistical guarantees of application response time in cloud settings. Cerebro combines off-line static analysis of application control structure with on-line cloud performance monitoring and statistical forecasting to predict bounds on the response time of web-facing application programming interfaces (APIs). Because Cerebro does not require application instrumentation or per-application cloud benchmarking, it does not impose any runtime overhead, and is suitable for use at cloud scales. Also, because the bounds are statistical, they are appropriate for use as the basis for SLAs between cloud-hosted applications and their users.

We investigate the correctness of Cerebro predictions, the tightness of their bounds, and the duration over which the bounds persist in both Google App Engine and AppScale (public and private cloud platforms respectively). We also detail the effectiveness of our SLA prediction methodology compared to other performance bound estimation methods based on simple statistical analysis.

***Keywords*** Cloud computing, Web APIs, SLA

## 1. Introduction

Web services, service oriented architectures, and cloud platforms have revolutionized the way developers engineer and deploy software. Using the web service model, developers create new applications by "mashing up" content and functionality from existing services exposed via web-accessible application programming interfaces (web APIs). This approach both expedites and simplifies implementation since developers can leverage the software engineering and maintenance efforts of others. Moreover, platform-as-a-service (PaaS) clouds for hosting web applications have emerged as a key technology for managing applications at scale in both public (managed) and private settings.

Consequently, web APIs are rapidly proliferating. At the time of this writing, ProgrammableWeb [35] indexes over 13,000 publicly available web APIs. These APIs increasingly employ the REST (Representational State Transfer) architectural style [13], and target both commercial (e.g. advertising, shopping, travel, etc.) and non-commercial (e.g. IEEE [22], UC Berkeley [5], US White House [44]) application domains.

Despite the many benefits, reusing existing services also has its costs. In particular, new applications become dependent on the services they compose. These dependencies impact correctness, performance, and availability of the composite applications – for which the "top level" developer is often held accountable. Compounding the situation, the underlying services can and do change over time while their APIs remain stable, unbeknownst to the developers that programmatically access them. Unfortunately, there is a dearth of tools that help developers reason about these dependencies throughout an application's lifecycle (i.e. development, deployment, and runtime). Without such tools, programmers must resort to extensive, continuous, and costly, testing and profiling to understand the performance impact on their applications that results from the increasingly complex collection of services that they depend on.

To address this issue, we present Cerebro, a new approach that predicts bounds on the response time performance of web APIs exported by applications that are hosted in a PaaS cloud. The goal of Cerebro is to allow a PaaS administrator to determine what response time service level agreement (SLA) can be fulfilled by each web API operation exported by the applications hosted in the PaaS.

SLAs typically specify a minimum service level, and a probability (usually large) that the minimum service level

*UCSB Technical Report 2015-004*,    April, 21 2014.

will be maintained. Cerebro uses a combination of static analysis of the hosted web APIs, and runtime monitoring of the PaaS cloud (*not* the web APIs themselves) to determine what minimum response time guarantee can be made with a target probability specified by a PaaS administrator. These calculated SLAs enable developers to reason about the performance of the applications that consume the cloud-hosted web APIs.

Currently, cloud computing systems such as Amazon Web Services (AWS) [3] and Google App Engine (GAE) [15] offer reliability SLAs specifying the fraction of availability over a fixed time period for their services that users can expect, when they contract to use a service. However, they do not provide SLAs guaranteeing minimum levels of performance. In contrast, Cerebro predictions make it possible to determine response time SLAs with probabilities specified by the cloud provider in a way that is scalable.

Cerebro is designed to improve the use of both public and private PaaS clouds which have emerged as popular application hosting venues [38]. For example, there are over four million active GAE applications that can execute on Google's public cloud or over AppScale, an open source, private cloud version of App Engine. A PaaS cloud provides developers with a collection of commonly used, scalable services, that the platform exports via APIs defined within a software development kit (cloud SDK). These services are fully managed and covered under the availability SLAs of the cloud platform. For example, services in GAE and AppScale cloud SDK include a distributed NoSQL datastore, task management, and data caching, among others.

Cerebro generates response time SLAs for API calls exported by a web application developed using the services available within the PaaS. For brevity, in this work we will use the term *web API* to refer to a web-accessible API exported by an application hosted on a PaaS platform. Further, we will use the term *cloud SDK* to refer to the APIs that are maintained as part of the PaaS and available to all hosted applications. This enables us to differentiate the internal APIs of the PaaS from the APIs exported by the deployed applications. For example, an application hosted in Google App Engine might export one or more web APIs to its users while leveraging the internal cloud SDK for the Google datastore that is available as part of the Google App Engine PaaS.

Cerebro uses static analysis to identify the cloud SDK invocations that dominate the response time of web APIs. Independently, Cerebro also maintains a running history of cloud SDK response time performance. It uses QBETS [33] – a forecasting methodology we have developed in prior work for predicting bounds on "ill behaved" univariate time series – to predict response time bounds on each cloud SDK invocation made by the application. It combines these predictions dynamically for each static program path through a web API operation, and returns the "worst-case" upper bound on the time necessary to complete the operation.

Because service implementations and platform behavior under load change over time, Cerebro's predictions necessarily have a lifetime. That is, the predicted SLAs may become invalid after some time. As part of this paper, we develop a model for detecting such SLA invalidations. We use this model to investigate the effective lifetime of Cerebro predictions. When such changes occur, Cerebro can be reinvoked to establish new SLAs for any deployed web API.

We have implemented Cerebro for both the Google App Engine public PaaS, and the AppScale private PaaS. Given its modular design and this experience, we believe that Cerebro can be easily integrated into any PaaS system. We use our prototype implementation to evaluate the accuracy of Cerebro as well as the tightness of the bounds it predicts (i.e. the difference between the predictions and the actual API execution times). To this end, we carry out a range of experiments using App Engine applications that are available as open source.

We also detail the duration over which these predictions hold in both GAE and AppScale. We find that Cerebro generates correct SLAs (predictions that meet or exceed their probabilistic guarantees), and that these SLAs are valid over time periods ranging from 1.4 hours to more than 24 hours. We also find that the high variability of performance in public PaaS clouds due to their multi-tenancy and massive scale requires that Cerebro be more conservative in its predictions to achieve the desired level of correctness. In comparison, Cerebro is able to make much tighter SLA predictions for web APIs hosted in private, single tenant clouds.

Because Cerebro provides this analysis statically it imposes no run-time overhead on the applications themselves. It requires no run-time instrumentation of application code, and it does not require any performance testing of the web APIs. Furthermore, because the PaaS is scalable and SDK monitoring data is shared across all Cerebro executions, the continuous monitoring of the cloud SDK generates no discernible load on the cloud platform. Thus we believe Cerebro is suitable for highly scalable cloud settings.

Finally, we have developed Cerebro for use with EAGER (**E**nforced **A**PI **G**overnance **E**ngine for **R**EST) [23] – an API governance system for PaaS clouds. EAGER attempts to enforce governance policies at the deployment-time of cloud applications. These governance policies are specified by cloud administrators to ensure the reliable operation of the cloud and the deployed applications. PaaS platforms include an application deployment phase during which the platform provisions resources for the application, installs the application components, and configures them to use the cloud SDKs. EAGER injects a policy checking and enforcement step into this deployment workflow so that only applications that are compliant with respect to site-specific policies are successfully deployed. Cerebro allows PaaS administrators to define EAGER policies that allow an application to be deployed *only* when its web APIs meet a predetermined (or pre-negotiated) SLA target, and to be notified by the platform when such SLAs require renegotiation.

We structure the rest of this paper as follows. We first characterize the domain of PaaS-hosted web APIs for GAE and AppScale in Section 2. We then present the design of Cerebro in Section 3 and overview our software architecture and prototype implementation. Next, we present our empirical evaluation of Cerebro in Section 4. Finally, we discuss related work (Section 5) and conclude (Section 6).

## 2. Domain Characteristics and Assumptions

The goal of our work is to analyze a web API statically, and from this analysis without deploying or running the web API, accurately predict an upper bound on its response time. With such a prediction, developers and cloud administrators can provide performance SLAs to the API consumers (human or programmatic), to help them reason about the performance implications of using APIs – something that is not possible today. For general purpose applications, such worst-case execution time analysis has been shown by numerous researchers to be challenging to achieve for all but simple programs or specific application domains.

To overcome these challenges, we take inspiration from the latter and exploit the application domain of PaaS-hosted web APIs to achieve our goal. In this paper, we focus on the popular Google App Engine (GAE) public PaaS and AppScale private PaaS, which support the same applications, development and deployment model, and platform services.

The first characteristic of PaaS systems that we exploit to facilitate our analysis is their predefined programming interfaces through which they export various platform services. Herein we refer to these programming interfaces as the cloud software development kit or the *cloud SDK*. We refer to the individual member interfaces of the cloud SDK as *cloud SDK interfaces*, and to their constituent operations as *cloud SDK operations*. These interfaces export scalable functionality that is commonly used to implement web APIs: key-value datastores, databases, data caching, task and user management, security and authentication, etc. The App Engine and AppScale cloud SDK is detailed in `https://cloud.google.com/appengine/docs/java/javadoc/`.

Figure 1 illustrates the PaaS development and deployment model. Developers implement their application code as a combination of calls to the cloud SDK and their own code. The service implementations for the cloud SDK are highly scalable, highly available (have SLAs associated with them), and automatically managed by the platform. Developers then upload their applications to the cloud for deployment. Once deployed, the applications and any web APIs exported by them can be accessed via HTTP/S requests by external or co-located clients.

Typically, PaaS-hosted web APIs perform one or more cloud SDK calls. The reason for this is two-fold. First, cloud SDKs provide web APIs with much of the functionality that they require. Second, PaaS clouds “sandbox” web APIs to

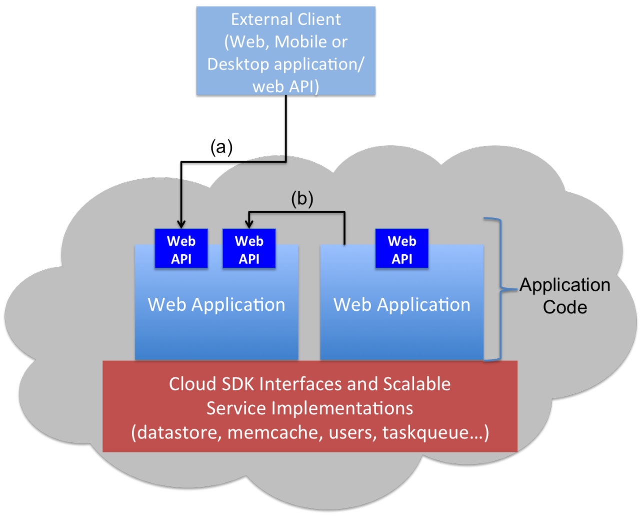
External Client (Web, Mobile or Desktop application/ web API)
(a)
(b)
Web API
Web API
Web API
Web Application
Web Application
Application Code
Cloud SDK Interfaces and Scalable Service Implementations (datastore, memcache, users, taskqueue...)

**Figure 1.** PaaS-hosted Web APIs: (a) An external client making requests to the API; (b) A PaaS-hosted web API invoking another in the same cloud.

enforce quotas, to enable billing, and to restrict certain functionality that can lead to security holes, platform instability, or scaling issues [16]. For example, GAE and AppScale cloud platforms restrict the application code from accessing the local file system, accessing shared memory, using certain libraries, and arbitrarily spawning threads. Therefore developers must use the provided cloud SDK operations to implement program logic equivalent to the restricted features. For example, the datastore interface can be used to read and write persistent data instead of using the local file system, and the memcache interface can be used in lieu of global shared memory.

Furthermore, the only way for a web API to execute is in response to an HTTP/S request or as a background task. Therefore, execution of all web API operations start and end at well defined program points, and we are able to infer this structure from common software patterns. Also, concurrency is restricted by capping the number of threads and requiring that a thread cannot outlive the request that creates it. Finally, PaaS clouds enforce quotas and limits on service (cloud SDK) use [16, 17, 28]. App Engine, for example, requires that all web API requests complete under 60 seconds. Otherwise they are terminated by the platform. Such enforcement places a strict upper bound on the execution of a web API operation.

To understand the specific characteristics of PaaS-hosted web APIs, and the potential of this restricted domain to facilitate efficient static analysis and response time prediction, we next summarize results from static analysis (using the Soot framework [43]) of 35 real world App Engine web APIs. These web APIs are open source, written in Java, and run over Google App Engine or AppScale without modification. We plan to make these applications publicly available upon publication.

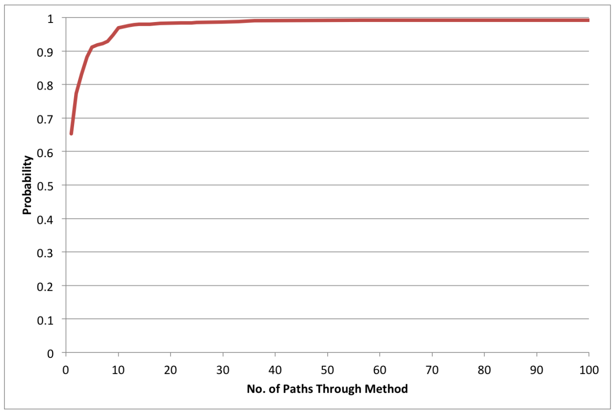

**Figure 2.** CDF of the number of static paths through methods in the surveyed web APIs.

Our analysis detected a total of 1458 Java methods in the analyzed codes. Figure 2 shows the cumulative distribution of static program paths in these methods. Approximately 97% of the methods considered in the analysis have 10 or fewer static program paths through them. 99% of the methods have 36 or fewer paths. However, the CDF is heavy tailed, and grows to 34992. We truncate the graph at 100 paths for clarity. As such, only a very small number of methods each contains a large number of paths. Fortunately, over 65% of the methods have exactly 1 path (i.e. there are no branches).

Next, we consider the looping behavior of web APIs. 1286 of the methods (88%) considered in the study do not have any loops. 172 methods (12%) contain loops. We believe that this characteristic is due to the fact that the PaaS SDK and the platform restrictions like quotas and response time limits discourage looping.

Approximately 29% of all the loops in the analyzed programs do not contain any cloud SDK invocations. A majority of the loops (61%) however, are used to iterate over a dataset that is returned from the datastore cloud SDK interface of App Engine (i.e iterating on the result set returned by a datastore query). We refer to this particular type of loops as *iterative datastore reads*.

Table 1 lists the number of times each cloud SDK interface is called across all paths and methods in the analyzed programs. The Datastore API is the most commonly used interface. This is because data management is fundamental to most web APIs and the PaaS disallows using the local filesystem to do so, for scalability and portability reasons.

Next, we explore the number of cloud SDK calls made along different paths of execution in the web APIs. For this study we consider all paths of execution through the methods (64780 total paths). Figure 3 shows the cumulative distribution of the number of SDK calls within paths. Approximately 98% of the paths have 1 cloud SDK call or fewer. The probability of finding an execution path with more than 5 cloud SDK calls is smaller than 1%.

**Table 1.** Static cloud SDK calls in surveyed web APIs

| Cloud SDK Interface | No. of Invocations |
|---|---|
| blobstore | 7 |
| channel | 1 |
| datastore | 735 |
| files | 4 |
| images | 3 |
| memcache | 12 |
| search | 6 |
| taskqueue | 24 |
| tools | 2 |
| urlfetch | 8 |
| users | 44 |
| xmpp | 3 |

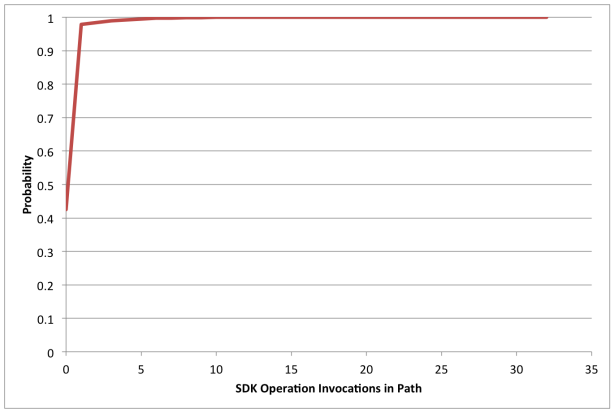

**Figure 3.** CDF of cloud SDK call counts in paths of execution.

Finally, our experience with App Engine web APIs indicates that a significant portion of the total time of a method (web API operation) is spent in cloud SDK calls. Confirming this hypothesis requires careful instrumentation (i.e. difficult to automate) of the web API codes. We performed such a test by hand on two representative applications and found that the time spent in code other than cloud SDK calls accounts for 0-6% of the total time (0-3ms for a 30-50ms web API operation).

This study of various characteristics typical of PaaS-hosted web APIs indicates that there may be opportunities to exploit the specific aspects of this application domain to simplify analysis and to facilitate performance prediction. In particular, operations in these applications are short, have a small number of paths to analyze, implement few loops, and invoke a small number of cloud SDK calls. Moreover, most of the time spent executing these operations results from cloud SDK invocations. In the next section, we describe our design and implementation of Cerebro that takes advantage of these characteristics and assumptions. We then use a Cerebro prototype to experimentally evaluate its efficacy for estimating the worst-case response time for applications from this domain.

## 3. Cerebro

Given the restricted application domain of PaaS-hosted web APIs, we believe that it is possible to design a system that predicts response time SLAs for them using only static information from the web API code itself. To enable this, we design Cerebro with three primary components:

- A static analysis tool that extracts sequences of cloud SDK operations for each path through a method (web API operation),
- A monitoring agent that runs in the target PaaS, and efficiently monitors the performance of the underlying cloud SDK operations, and
- An SLA predictor that uses the outputs of these two components to accurately predict an upper bound on the response time of the web API.

We overview each of these components in the subsections that follow, and then discuss the Cerebro workflow with an example.

### 3.1 Static Analysis

This component analyzes the source code of the web API (or some intermediate representation of it) and extracts a sequence of cloud SDK operations. We implement our analysis for Java bytecode programs using the Soot framework [43]. Currently, our prototype analyzer considers the following Java codes as exposed web APIs.

- classes that extend the *javax.servlet.HttpServlet* class (i.e. Java servlet implementations)
- classes that contain JAX-RS @*Path* annotations, and
- any other classes explicitly specified by the developer in a special configuration file.

Cerebro performs a simple construction and interprocedural static analysis of control flow graph (CFG) [1, 2, 29, 30] for each web API operation. The algorithm extracts all cloud SDK operations along each path through the methods. Cerebro analyzes other functions that the method calls, recursively. Cerebro caches cloud SDK details for each function once analyzed so that it can be reused efficiently for other call sites to the same function. Cerebro does not analyze third-party library calls, if any (which in our experience typically do not contain cloud SDK calls). Cerebro encodes each cloud SDK call sequence for each path in a lookup table. We identify cloud SDK calls by their Java package name (e.g. `com.google.appengine.apis`).

To handle loops, we first extract them from the CFG and annotate all cloud SDK invocations that occur within them. We annotate each such SDK invocation with an estimate on the number of times the loop is likely to execute in the worst case. We estimate loop bounds using a loop bound prediction algorithm based on abstract interpretation [8].

As shown in the previous section, loops in these programs are rare and, when they do occur, they are used to iterate over a dataset returned from a database. For such data-dependent loops, we estimate the bounds if specified in the cloud SDK call (e.g. the maximum number of entities to return [18]). If our analysis is unable to estimate the bounds for these loops, Cerebro prompts the developer for an estimate of the likely dataset size and/or loop bounds.

### 3.2 PaaS Monitoring Agent

Cerebro monitors and records the response time of individual cloud SDK operations within a running PaaS system. Such support can be implemented as a PaaS-native feature or as a PaaS application (web API); we use the latter in our prototype. The monitoring agent runs in the background with, but separate from, other PaaS-hosted web APIs. The agent invokes cloud SDK operations periodically on synthetic datasets and records timestamped response times in the PaaS datastore for each cloud SDK operation. Finally, the agent periodically reclaims old measurement data to eliminate unnecessary storage. The Cerebro monitoring and reclamation rates are configurable, and monitoring benchmarks can be added and customized easily to capture common PaaS-hosted web API coding patterns.

In our prototype, the agent monitors the datastore and memcache SDK interfaces every 60 seconds. In addition, it benchmarks loop iteration over datastore entities to capture the performance of iterative datastore reads for datastore result set sizes of 10, 100, and 1000. We limit ourselves to these values because the PaaS requires that all operations complete (respond) within 60 seconds – so the data sizes returned are typically small.

### 3.3 Making SLA Predictions

To make SLA predictions, Cerebro uses Queue Bounds Estimation from Time Series (QBETS) [33], a non-parametric time series analysis method that we developed in prior work. We originally designed QBETS for predicting the scheduling delays for the batch queue systems used in high performance computing environments but it has proved effective in other settings where forecasts from arbitrary times series are needed [7, 32, 46]. In particular, it is both non-parametric and it automatically adapts to changes in the underlying time series dynamics making it useful in settings where forecasts are required from arbitrary data with widely varying characteristics.

A QBETS analysis requires three inputs:

1. A time series of data generated by a continuous experiment.
2. The percentile for which an upper bound should be predicted ($p \in [1..99]$).
3. The upper confidence level of the prediction ($c \in (0,1)$).

QBETS uses this information to predict an upper bound for the $p^{th}$ percentile of the time series. It does so by treating each observation in the time series as a Bernoulli trial with probability $0.01p$ of success. Let $q = 0.01p$. If there are $n$ observations, the probability of there being exactly $k$ successes is described by a Binomial distribution (assuming observation independence) having parameters $n$ and $q$. If $Q$ is the $p^{th}$ percentile of the distribution from which the observations have been drawn, the equation

$$1 - \sum_{j=0}^{k} \binom{n}{j} \cdot (1-q)^j \cdot q^{n-j} \tag{1}$$

gives the probability that more than $k$ observations are greater than $Q$. As a result, the $k^{th}$ largest value in a sorted list of $n$ observations gives an upper $c$ confidence bound on $Q$ when $k$ is the smallest integer value for which Equation 1 is larger than $c$.

More succinctly, QBETS sorts observations in a history of observations, and computes the value of $k$ that constitutes an index into this sorted list that is the upper $c$ confidence bound on the $p^{th}$ percentile. The methodology assumes that the time series of observations is ergodic so that, in the long run, the confidence bounds are accurate.

QBETS also attempts to detect change points in the time series of observations so that it can apply this inference technique to only the most recent segment of the series that appears to be stationary. To do so, it compares percentile bounds predictions with observations throughout the series and determines where the series is likely to have undergone a change. It then discards observations from the series prior to this change point and continues. As a result, when QBETS starts, it must "learn" the series by scanning it in time series order to determine the change points. We report Cerebro learning time in our empirical evaluation in Subsection 4.4.

Note that $c$ is an upper confidence level on $p^{th}$ percentile which makes the QBETS bound estimates conservative. That is, the value returned by QBETS as a bound prediction is larger than the true $p^{th}$ percentile with probability $1-c$ under the assumptions of the QBETS model. In this study, we use the $95^{th}$ percentile and $c = 0.01$.

Note that the algorithm itself can be implemented efficiently so that it is suitable for on-line use. Details of this implementation as well as a fuller accounting of the statistical properties and assumptions are available in [7, 31–33].

QBETS requires a sufficiently large number of data points in the input time series before it can make an accurate prediction. Specifically, the largest value in a sorted list of $n$ observations is greater than the $p^{th}$ percentile with confidence $c$ when $n >= log(c)/log(0.01p)$.

For example, predicting the $95^{th}$ percentile of the API execution time, with an upper confidence of 0.01 requires at least 90 observations. We use this information to control reclamation of monitoring data by PaaS agent.

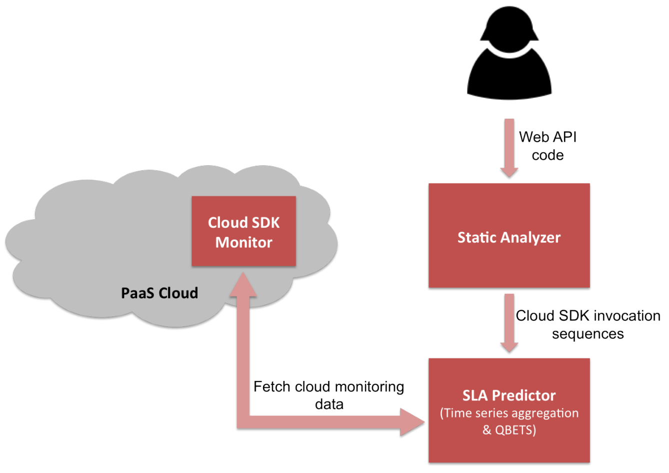
Web API code
Static Analyzer
Cloud SDK invocation sequences
Cloud SDK Monitor
PaaS Cloud
Fetch cloud monitoring data
SLA Predictor
(Time series aggregation & QBETS)

**Figure 4.** Cerebro architecture and component interactions.

## 3.4 Example Cerebro Workflow

Figure 4 illustrates how the Cerebro components interact with each other during the prediction making process. Cerebro can be invoked when a web API is deployed to a PaaS cloud or at any time during the development process to give developers insight into the worst-case response time of their applications.

Upon invoking Cerebro with a web API code, Cerebro performs its static analysis on all operations in the API. For each analyzed operation it produces a list of annotated cloud SDK invocation sequences – one sequence per program path. Cerebro then prunes this list to eliminate duplicates. Duplicates occur when a web API operation has multiple program paths with the same sequence of cloud SDK invocations. Next, for each pruned list Cerebro performs the following operations:

1. Retrieve (possibly compressed) benchmarking data from the monitoring agent for all SDK operations in each sequence. The agent returns ordered time series data (one time series per cloud SDK operation).
2. Align retrieved time series across operations in time, and sum the aligned values to form a single *joint* time series of the summed values for the sequence of cloud SDK operations.
3. Run QBETS on the joint time series with the desired $p$ and $c$ values to predict an upper bound.

Cerebro uses largest predicted value (across path sequences) as its SLA prediction for a web API operation. This process (SLA prediction) can be implemented as a co-located service in the PaaS cloud or as a standalone utility. We do the latter in our prototype.

As an example, suppose that the static analysis results in the cloud SDK invocation sequence $< op_1, op_2, op_3 >$ for some operation in a web API. Assume that the monitoring agent has collected the following time series for the three SDK operations:

- $op_1$: $[t_0 : 5, t_1 : 4, t_2 : 6, ...., t_n : 5]$
- $op_2$: $[t_0 : 22, t_1 : 20, t_2 : 21, ...., t_n : 21]$
- $op_3$: $[t_0 : 7, t_1 : 7, t_2 : 8, ...., t_n : 7]$

Here $t_m$ is the time at which the $m^{th}$ measurement is taken. Cerebro aligns the three time series according to timestamps, and sums the values to obtain the following joint time series: $[t_0 : 34, t_1 : 31, t_2 : 35, ...., t_n : 33]$

If any operation is tagged as being inside a loop, where the loop bounds have been estimated, Cerebro multiplies the time series data corresponding to that operation by the loop bound estimate before aggregating. In cases where the operation is inside a data-dependent loop, we request the time series data from the monitoring agent for its iterative datastore read benchmark for a number of entities that is equal to or larger than the annotation and include it in the joint time series.

Cerebro passes the final joint time series for each sequence of operations to QBETS, which returns the worst-case upper bound response time it predicts. If the QBETS predicted value is $Q$ milliseconds, Cerebro forms the SLA as "the web API will respond in under $Q$ milliseconds, $p\%$ of the time". When the web API has multiple operations, Cerebro estimates multiple SLAs for the API. If a single value is needed for the entire API regardless of operation, Cerebro returns the largest predicted value as the final SLA (i.e. the worst-case SLA for the API).

## 4. Experimental Results

To empirically evaluate Cerebro, we conduct experiments using five open source, Google App Engine applications.

**StudentInfo** RESTful (JAX-RS) application for managing students of a class (adding, removing, and listing student information).

**ServerHealth** Monitors, computes, and reports statistics for server uptime for a given web URL.

**SocialMapper** A simple social networking application with APIs for adding users and comments.

**StockTrader** A stock trading application that provides APIs for adding users, registering companies, buying and selling stocks among users.

**Rooms** A hotel booking application with APIs for registering hotels and querying available rooms.

These web APIs use the datastore cloud SDK interface extensively. The Rooms web API also uses the memcache interface. We focus on these two interfaces exclusively in this study. We execute these applications in the Google App Engine public cloud (SDK v1.9.17) and in an AppScale (v2.0) private cloud. We instrument the programs to collect execution time statistics for verification purposes only (the instrumentation data is not used to predict the SLAs). The AppScale private cloud used for testing was hosted using four "m3.2xlarge" virtual machines running on a private Eucalyptus [34] cloud.

We first report the time required for Cerebro to perform its analysis and SLA prediction. Across web APIs, Cerebro takes 10.00 seconds on average, with a maximum time of 14.25 seconds for StudentInfo application. These times include the time taken by the static analyzer to analyze all the web API operations and the time taken by QBETS to make predictions. For these results, the length of the time series collected by PaaS monitoring agent is 1528 data points (25.5 hours of monitoring data). Since the QBETS analysis time depends on the length of the input time series, we also measured the time for 2 weeks of monitoring data (19322 data points) to provide some insight into the overhead of SLA prediction. Even in this case, Cerebro requires only 574.05 seconds (9.6 minutes) on average.

### 4.1 Correctness of Predictions

We first evaluate the correctness of Cerebro predictions. A set of predictions is *correct* if the *fraction* of measured response time values that fall below the Cerebro prediction is greater than or equal to the SLA target probability. For example, if the SLA probability is 0.95 (i.e. $p = 95$ in QBETS) for a specific web API, then the Cerebro predictions are correct if at least 95% of the response times measured for the web API are smaller than their corresponding Cerebro predictions.

We benchmark each web API for a period of 15 to 20 hours. During this time we run a remote HTTP client that makes requests to the web APIs once every minute. The application instrumentation measures and records the response time of the API operation for each request (i.e. within the application). Concurrently and within the same PaaS system, we execute the Cerebro PaaS monitoring agent which is an independently hosted application within the cloud that benchmarks each SDK operation once every minute.

Cerebro predicts the web API execution times using only the cloud SDK benchmarking data collected by Cerebro's PaaS monitoring agent. We configure Cerebro to predict an upper bound for the $95^{th}$ percentile of the web API response time, with an upper confidence of 0.01.

QBETS generates a prediction for *every* value in the input time series (one per minute). Cerebro reports the last one as the SLA prediction to the user or PaaS administrator in production. However, having per-minute predictions enables us to compare these predictions against actual web API execution times measured during the same time period to evaluate Cerebro correctness. More specifically, we associate with each measurement the prediction from the prediction time series that most nearly precedes it in time. The correctness fraction is computed from a sample of 1000 prediction-measurement pairs.

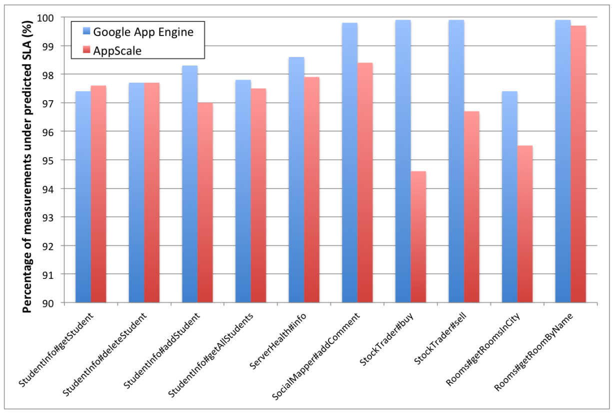

**Figure 5.** Cerebro correctness percentage in Google App Engine and AppScale cloud platforms.

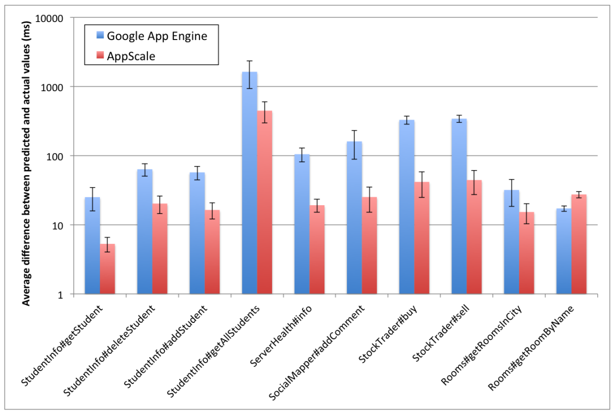

**Figure 6.** Average difference between predictions and actual response times in Google App Engine and AppScale. The y-axis is in log scale.

Figure 5 shows the final results of this experiment. Each of the columns in Figure 5 corresponds to a single web API operation in one of the sample applications. The columns are labeled in the form of *ApplicationName#OperationName*, a convention we will continue to use in the rest of the paper.

Since we are using Cerebro to predict the 95th percentile of the API response times, Cerebro's predictions are correct when at least 95% of the measured response times are less than their corresponding predicted upper bounds. According to Figure 5, Cerebro achieves this goal for all the applications in both cloud environments.

The web API operations illustrated in Figure 5 cover a wide spectrum of scenarios that may be encountered in real world. StudentInfo#getStudent and StudentInfo#addStudent are by far the simplest operations in the mix. They invoke a single cloud SDK operation each, and perform a simple datastore read and a simple datastore write respectively. As per our survey results, these alone cover a significant portion of the web APIs developed for the App Engine and AppScale cloud platforms (1 path through the code, and 1 cloud SDK call). The StudentInfo#deleteStudent operation makes two cloud SDK operations in sequence, whereas StudentInfo#getAllStudents performs an iterative datastore read. In our experiment with StudentInfo#getAllStudents, we had the datastore preloaded with 1000 student records, and Cerebro was configured to use a maximum entity count of 1000 when making predictions.

ServerHealth#info invokes the same cloud SDK operation three times in sequence. Both StockTrader#buy and StockTrader#sell have multiple paths through the application (due to branching), thus causing Cerebro to make multiple sequences of predictions – one sequence per path. The results shown in Figure 5 are for the longest paths which consist of seven cloud SDK invocations each. According to our survey, 99.8% of the execution paths found in Google App Engine applications have seven or fewer cloud SDK calls in them. Therefore we believe that the StockTrader web API represents an important upper bound case.

Rooms#getRoomByName invokes two different cloud SDK interfaces, namely datastore and memcache. Rooms#getAllRooms is another operation that consists of an iterative datastore read. In this case, we had the datastore preloaded with 10 entities, and Cerebro was configured to use a maximum entity count of 10.

## 4.2 Tightness of Predictions

In this section we discuss the tightness of the predictions generated by Cerebro. Tightness is a measure of how closely the predictions bound the actual response times of the web APIs. Note that it is possible to perfectly achieve the correctness goal by simply predicting overly large values for web API response times. For example, if Cerebro were to predict a response time of several years for exactly 95% of the web API invocations and zero for the others, it would likely achieve a correctness percentage of 95%. From a practical perspective, however, such an extreme upper bound is not useful as the basis for an SLA.

Figure 6 depicts the average difference between predicted response time bounds and actual response times for our sample web APIs when running in the App Engine and AppScale clouds. These results were obtained considering a sequence of 1000 consecutive predictions (of 95th percentile) and the averages are computed only for correct predictions (i.e. ones above their corresponding measurements).

According to Figure 6, Cerebro generates fairly tight SLA predictions for most web API operations considered in the experiments. In fact, 14 out of the 20 cases illustrated in the figure show average difference values less than 65ms. In a few cases, however, the bounds differ from the average measurement substantially:

- StudentInfo#getAllStudents on both cloud platforms
- ServerHealth#info, SocialMapper#addComment, StockTrader#buy and StockTrader#sell on App Engine

Figure 7 shows the empirical cumulative distribution function (CDF) of measured execution times for the StudentInfo#getAllStudents on Google App Engine (one of the

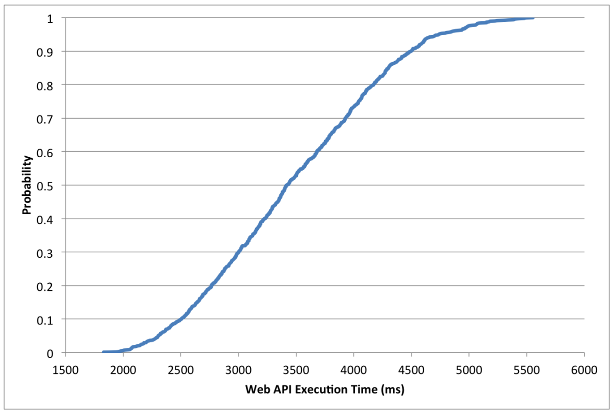

**Figure 7.** CDF of measured executions times of the StudentInfo#getAllStudents operation on App Engine.

extreme cases). This distribution was obtained by considering the application's instrumentation results gathered within a window of 1000 minutes. The average of this sample is 3431.79ms, and the $95^{th}$ percentile from the CDF is 4739ms. Thus, taken as a distribution, the "spread" between the average and the $95^{th}$ percentile is more than 1300ms.

From this, it becomes evident that StudentInfo#getAllStudents records very high execution times frequently. In order to incorporate such high outliers, Cerebro must be conservative and predict large values for the $95^{th}$ percentile. This is a required feature to ensure that 95% or more API invocations have execution times under the predicted SLA. But as a consequence, the average distance between the measurements and the predictions increases significantly.

We omit a similar analysis of the other cases in the interest of brevity but summarize the tightness results as indicating that Cerebro achieves a bound that is "tight" with respect to the percentiles observed by sampling the series for long periods.

Another interesting observation we can make regarding the tightness of predictions is that the predictions made in the AppScale cloud platform are significantly tighter than the ones made in Google App Engine (Figure 6). For nine out of the ten operations tested, Cerebro has generated tighter predictions in the AppScale environment. This is because web API performance on AppScale is far more stable and predictable thus resulting in fewer measurements that occur far from the average.

The reason why AppScale's performance is more stable over time is because it is deployed on a set of closely controlled and monitored cluster of virtual machines (VMs) that use a private Infrastructure-as-a-Service (IaaS) cloud to implement isolation. In particular, the VMs assigned to AppScale do not share nodes with "noisy neighbors" in our test environment. In contrast, Google App Engine does not expose the performance characteristics of its multi-tenancy. While it operates at vastly greater scale, our test applications also exhibit wider variance of web API response time when using it. Cerebro, however, is able to predict a correct and tight SLAs for applications running in either platform: the lower variance private AppScale PaaS, and the extreme scale but more varying Google App Engine PaaS.

## 4.3 Duration of Prediction Validity

To be of practical value to PaaS administration, the duration over which a Cerebro prediction remains valid must be long enough to allow appropriate remedial action when load conditions change, and the SLA is in danger of being violated. In particular, SLAs must remain correct for at least the time necessary to allow human responses to changing conditions such as the commitment of more resources to web APIs that are in violation or alerts to support staff that customers may be calling to claim SLA breach. Ideally, each prediction should persist as correct for several hours or more to match staff response time to potential SLA violations.

However, determining when a Cerebro-predicted SLA becomes invalid is potentially complex. For example, given the definition of correctness described in Subsection 4.1, it is possible to report an SLA violation when the running tabulation of correctness percentage falls below the target probability (when expressed as a percentage). However, if this metric is used, and Cerebro is correct for many consecutive measurements, a sudden change in conditions that causes the response time to persist at a higher level will not immediately trigger a violation. For example, Cerebro might be correct for several consecutive months and then incorrect for several consecutive weeks before the overall correctness percentage drops below 95% and a violation is detected. If the SLA is measured over a year, such time scales may be acceptable but we believe that PaaS administrators would consider such a long period of time where the SLAs were continuously in violation unacceptable. Thus we propose a more conservative approach to measuring the duration over which a prediction remains valid than simply measuring the time until the correctness percentage drops below the SLA-specified value.

Suppose at time $t$ Cerebro predicts value $Q$ as the $p$-th percentile of some API's execution time. If $Q$ is a correct and tight prediction, the probability of API's next measured response time being greater than $Q$ is $1-(0.01p)$. If the time series consists of independent measurements then the probability of seeing $n$ consecutive values greater than $Q$ (due to random chance) is $(1-0.01p)^n$. For example, using the $95^{th}$ percentile, the probability of seeing 3 values in a row larger than the actual percentile (purely due to random chance) is $(0.05)^3 = 0.00012$ or about 1 in 8333.

This calculation is conservative with respect to autocorrelation. That is, if the time series is stationary but autocorrelated, then the number of consecutive values above the $95^{th}$ percentile that correspond to a probability of 0.00012 is larger than 3. For example, in previous work [33] using an artificially generated AR(1) series, we observed that 5 consecutive values above the $95^{th}$ percentile occurred with probability 0.00012 when the first autocorrelation was 0.5,

**Table 2.** Prediction validity period distributions of different operations in App Engine. Validity durations were computed by observing 3 consecutive SLA violations. $5^{th}$ and $95^{th}$ columns represent the 5th and 95th percentiles of the distributions respectively. All values are in hours.

| Operation | $5^{th}$ | Average | $95^{th}$ |
|---|---|---|---|
| StudentInfo#getStudent | 7.15 | 70.72 | 134.43 |
| StudentInfo#deleteStudent | 2.55 | 37.97 | 94.37 |
| StudentInfo#addStudent | 1.45 | 26.8 | 64.78 |
| ServerHealth#info | 1.41 | 39.22 | 117.71 |
| Rooms#getRoomByName | 7.24 | 70.47 | 133.36 |
| Rooms#getRoomsInCity | 2.08 | 30.12 | 82.58 |

**Table 3.** Prediction validity period distributions of different operations in AppScale. Validity periods were computed by observing 3 consecutive SLA violations. $5^{th}$ and $95^{th}$ columns represent the 5th and 95th percentiles of the distributions respectively. All values are in hours.

| Operation | $5^{th}$ | Average | $95^{th}$ |
|---|---|---|---|
| StudentInfo#getStudent | 6.1 | 60.67 | 115.24 |
| StudentInfo#deleteStudent | 6.08 | 60.21 | 114.32 |
| StudentInfo#addStudent | 6.1 | 60.67 | 115.24 |
| ServerHealth#info | 6.29 | 54.53 | 108.14 |
| Rooms#getRoomByName | 6.07 | 59.18 | 112.28 |
| Rooms#getRoomsInCity | 1.95 | 33.77 | 84.63 |

and 14 when the first autocorrelation was 0.85. QBETS uses a look-up table of these values to determine the number of consecutive measurements above $Q$ that constitute a "rare event" indicating a possible change in conditions.

Each time Cerebro makes a new prediction, it computes the current autocorrelation and uses the QBETS rare-event look-up table to determine $n$: the number of consecutive values that constitute a rare event. We measure the time from when Cerebro makes the prediction until we observe $n$ consecutive measurement values above that prediction as being the time duration over which the prediction is valid. We refer to this duration as the *validity duration*. Tables 2 and 3 present these durations for Cerebro predictions in Google App Engine and AppScale respectively.

From Table 2 the average validity duration for all 6 operations considered in App Engine is longer than 24 hours. The lowest average value observed is 26.8 hours, and that is for the StudentInfo#addStudent operation. If we just consider the $5^{th}$ percentiles of the distributions, they are also longer than 1 hour. The smallest $5^{th}$ percentile value of 1.41 hours is given by the ServerHealth#info operation. This result implies that, based on our conservative model for detecting SLA violations, Cerebro predictions made on Google App Engine would be valid for at least 1.41 hours or more, at least 95% of the time.

By comparing the distributions for different operations we can conclude that API operations that perform a single basic datastore or memcache read tend to have longer validity durations. In other words, those cloud SDK operations have fairly stable performance characteristics in Google App Engine. This is reflected in the $5^{th}$ percentiles of StudentInfo#getStudent and Rooms#getRoomByName. Alternatively operations that execute writes, iterative reads or long sequences of cloud SDK operations have shorter prediction validity durations.

For AppScale, the smallest average validity period of 33.77 hours is observed from the Rooms#getRoomsInCity operation. All other operations tested in AppScale have average prediction validity periods greater than 54 hours. The lowest $5^{th}$ percentile value in the distributions, which is 1.95 hours, is also shown by Rooms#getRoomsInCity. This means, the SLAs predicted for AppScale would hold correct for at least 1.95 hours or more, at least 95% of the time. The relatively smaller validity period values computed for the Rooms#getRoomsInCity operation indicates that the performance of iterative datastore reads is subject to some variability in AppScale.

## 4.4 Effectiveness of QBETS

In order to gauge the effectiveness of QBETS, we compare it to a "naïve" approach that simply uses the running empirical percentile tabulation of a given joint time series as a prediction. This *simple predictor* retains a sorted list of previous observations and predicts the $p$-th percentile to be the value that is larger than $p\%$ of the values in the observation history. Whenever a new observation is available, it is added to the history and each prediction uses the full history.

Figure 8 shows the correctness measurements for the simple predictor using the same cloud SDK monitoring data and application benchmarking data that was used in Subsection 4.1. That is, we keep the rest of Cerebro unchanged, swap QBETS out for the simple predictor, and run the same set of experiments using the logged observations. Thus the results in Figure 8 are directly comparable to Figure 5 where Cerebro uses QBETS as a forecaster.

For the simple predictor, Figure 8 shows lower correctness percentages compared to Figure 5 for QBETS (i.e. the simple predictor is less conservative). However, in several cases the simple predictor falls well short of the target correctness of 95% necessary for the SLA. That is, it is unable to furnish a prediction correctness that can be used as the basis of an SLA in all of the test cases.

To illustrate why the simple predictor fails to meet the desired correctness level, Figure 9 shows the time series of observations, simple predictor forecasts, and QBETS forecasts for the Rooms#getRoomsInCity operation on Google App Engine (the case in Figure 8 that shows lowest correctness percentage).

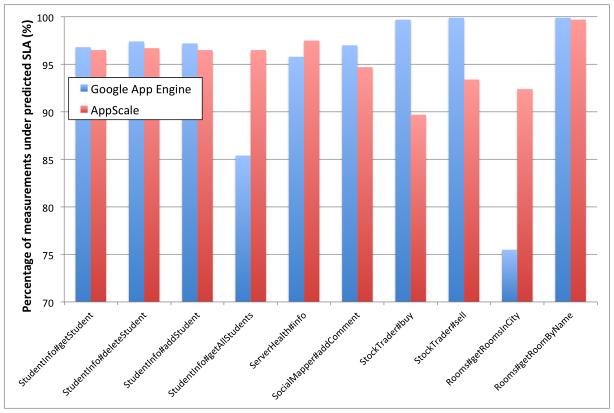

**Figure 8.** Cerebro correctness percentage resulting from the simple predictor (without QBETS).

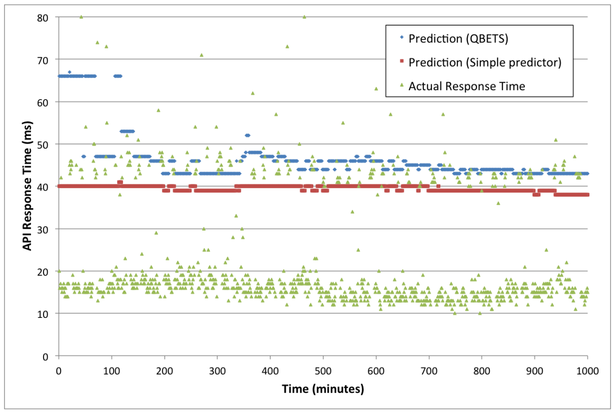

**Figure 9.** Comparison of predicted and actual response times of Rooms#getRoomsInCity on Google App Engine.

In this experiment, there are a significant number of response time measurements that violate the SLA given by simple predictor (i.e. are larger than the predicted percentile), but are below the corresponding QBETS prediction made for the same observation. Notice also that while QBETS is more conservative (its predictions are generally larger than those made by the simple predictor), in this case the predictions are typically only 10% larger. That is, while the simple predictor shows the $95^{th}$ percentile to be approximately $40ms$, the QBETS predictions vary between $42ms$ and $48ms$, except at the beginning where QBETS is "learning" the series. This difference in prediction, however, results in a large difference in correctness percentage. For QBETS, the correctness percentage is 97.4% (Figure 5) compared to 75.5% for the simple predictor (Figure 8).

## 5. Related Work

Our research leverages a number of mature research areas in computer science and mathematics. These areas include static program analysis, cloud computing, time series analysis, and SOA governance.

The problem of predicting execution time SLAs of web APIs is similar to worst-case execution time (WCET) analysis [12, 14, 30, 37, 45]. The objective of WCET analysis is to determine the maximum execution time of a software component in a given hardware platform. It is typically discussed in the context of real-time systems, where the developers should be able to document and enforce precise hard real-time constraints on the execution time of programs. In order to save time, manpower and hardware resources, WCET analysis solutions are generally designed favoring static analysis methods over software testing. We share similar concerns with regard to cloud platforms, and strive to eliminate software testing in the favor of static analysis.

Ermedahl et al designed and implemented SWEET [12], a WCET analysis tool that make use of program slicing [37], abstract interpretation [10] and invariant analysis [30] to determine the loop bounds and worst-case execution time of a program. Program slicing helps to reduce the amount of code and program states that need to be analyzed by SWEET. This is similar to our idea of extracting just the cloud SDK invocations form a given web API code. SWEET uses abstract interpretation in interval and congruence domains to identify the set of values that can be assigned to key control variables of a program. These sets are then used to calculate exact loop bounds for most data-independent loops in the code. Invariant analysis is used to detect variables that do not change during the course of a loop iteration, and remove them from the analysis thus further simplifying the loop bound estimation. Lokuceijewski et al propose a similar WCET analysis using program slicing and abstract interpretation [25]. They additionally use a technique called polytope models to speed up the analysis.

The corpus of research that covers the use of static analysis methods to estimate the execution time of software applications is extensive. Gulwani, Jain and Koskinen used two techniques named control-flow refinement and progress invariants to estimate the bounds for procedures with nested and multi-path loops [19]. Gulwani, Mehra and Chilimbi proposed SPEED [20], a system that computes symbolic bounds for programs. This system makes use of user-defined quantitative functions to predict the bounds for loops iterating over data structures like lists, trees and vectors. Our idea of using user-defined values to bound data-dependent loops (e.g. iterative datastore reads) is partly inspired by this concept. Bygde [8] proposed a set of algorithms for predicting data-independent loops using abstract interpretation and element counting (a technique that was partly used in [12]). Cerebro incorporates minor variations of these algorithms successfully due to their simplicity.

Cerebro makes use of and is similar to many of the execution time analysis systems discussed above. However, there are also several key differences. For instance, Cerebro is focused on solving the execution time prediction problem for PaaS-hosted web APIs. As we show in our characterization survey, such applications have a set of unique properties, that can be used to greatly simplify static analysis. Also, Cerebro is designed to only work with web API codes. This makes designing a solution much more simpler but less general. To handle the highly variable and evolving nature of cloud platforms, Cerebro combines static analysis with runtime monitoring of cloud platforms at the level of SDK operations. No other system provides such a hybrid approach to the best of our knowledge. Finally, we use time series analysis [33] to predict API execution time upper bounds with specific confidence levels.

SLA management on service-oriented systems and cloud systems has been throughly researched over the years. However, a lot of the existing work has focused on issues such as SLA monitoring [6, 27, 36, 42], SLA negotiation [26, 47, 48], and SLA modeling [9, 39, 40]. Some work has looked at incorporating a given SLA to the design of a system, and then monitoring it at the runtime to ensure SLA compliant behavior [21]. Our research takes a different approach from such works, whereby it attempts to predict the performance SLAs for a given web API. To the best of our knowledge, Cerebro is the first system to predict performance SLAs for web APIs developed for PaaS clouds.

A work that is similar to ours has been proposed by Ardagna, Damiani and Sagbo in [4]. The authors develop a system for early estimation of service performance based on simulations. Given a STS model (Symbolic Transitions System) of a service, their system is able to generate a simulation script, which can be used to assess the performance of the service. STS models are a type of finite state automata. Further, they use probabilistic distributions with fixed parameters to represent the delays incurred by various operations in the service. Cerebro is easier to use than this system because we do not require API developers to construct any models of the web APIs. Also, instead of using probabilistic distributions with fixed parameters, Cerebro uses actual historical performance metrics of cloud SDK operations. This enables Cerebro to generate more accurate results, that reflect the dynamic nature of the cloud platform.

There has also been prior work in the area of predicting SLA violations [11, 24, 41]. These systems take an existing SLA and historical performance data of a service, and predict when the service might violate the given SLA in the future. Cerebro's notion of prediction validity period has some relation to this line of research. However, Cerebro's main goal is to make SLA predictions for web APIs *before* they are deployed and executed. We believe that some of these existing SLA violation predictors can complement our work by providing API developers and cloud administrators insights on when a Cerebro-predicted SLA will be violated.

## 6. Conclusions and Future Work

Web services and service oriented architecture encourage developers to create new applications by composing existing services via their web APIs. But such integration of services makes it difficult to reason about the performance and other non-functional properties of composite applications. To facilitate such reasoning, web APIs should be exported for use by applications with strong performance guarantees (SLAs).

To this end, we propose Cerebro, a system that predicts response time bounds for web APIs deployed in PaaS clouds. We choose PaaS clouds as the target environment of our research due to their rapidly growing popularity as a technology for hosting scalable web APIs, and their SDK-based, restricted application development model, which makes it easier to analyze PaaS applications statically.

Cerebro uses static analysis to extract the sequence of cloud SDK calls made by a given web API code combined with historical performance measurements of cloud SDK calls to predict the response time of the web API. It employs QBETS, a non-parametric time series analysis and forecasting method, to analyze cloud SDK performance data, and predict bounds on response time that can be used as statistical "guarantees" with associated guarantee probabilities. Cerebro is intended for use both during development and deployment phases of a web API, and precludes the need for continuous performance testing of the API code. Further, it does not interfere with the run-time operation (i.e. it requires no application instrumentation at runtime) which makes it scalable.

We have implemented a prototype of Cerebro for Google App Engine public PaaS and AppScale private PaaS and evaluate it using a set of representative and open source web applications developed by others. Our findings indicate that the prototype can determine response time levels that correspond to specific target SLAs. These predictions are also durable, with average validity times varying between one and three days.

In the current design, Cerebro's cloud SDK monitoring agent only monitors a predefined set of cloud SDK operations. In our future work we wish to explore the possibility of making this component more dynamic, so that it automatically learns what operations to benchmark from the web APIs deployed in the cloud. We also plan to investigate further how to better handle data-dependent loops (iterative datastore reads) for different workloads. Further, we plan to integrate Cerebro with EAGER, our API governance system and policy engine for PaaS clouds, so that PaaS administrators can enforce SLA-related policies on web APIs at deployment-time.

## Acknowledgments

This work is funded in part by NSF (CNS-0905237, CNS-1218808, ACI-0751315) & NIH (1R01EB014877-01).

## References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-10088-6.

[2] F. E. Allen. Control Flow Analysis. In *Symposium on Compiler Optimization*, 1970.

[3] Amazon AWS. Amazon Web Services home page, 2015. `http://aws.amazon.com/` [Accessed March 2015].

[4] C. Ardagna, E. Damiani, and K. Sagbo. Early Assessment of Service Performance Based on Simulation. In *IEEE International Conference on Services Computing (SCC)*, 2013.

[5] Berkeley API Central. Berkeley API Central, 2015. `https://developer.berkeley.edu` [Accessed March 2015].

[6] A. Bertolino, G. De Angelis, A. Sabetta, and S. Elbaum. Scaling Up SLA Monitoring in Pervasive Environments. In *International Workshop on Engineering of Software Services for Pervasive Environments*, 2007.

[7] J. Brevik, D. Nurmi, and R. Wolski. Quantifying Machine Availability in Networked and Desktop Grid Systems. In *Proceedings of CCGrid04*, April 2004.

[8] S. Bygde. *Static WCET analysis based on abstract interpretation and counting of elements*. PhD thesis, Mälardalen University, 2010.

[9] T. Chau, V. Muthusamy, H.-A. Jacobsen, E. Litani, A. Chan, and P. Coulthard. Automating SLA Modeling. In *Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, 2008.

[10] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1977.

[11] S. Duan and S. Babu. Proactive Identification of Performance Problems. In *ACM SIGMOD International Conference on Management of Data*, 2006.

[12] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation, and Invariant Analysis. In *WCET*, 2007.

[13] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. AAI9980887.

[14] C. Frost, C. S. Jensen, K. S. Luckow, and B. Thomsen. WCET Analysis of Java Bytecode Featuring Common Execution Environments. In *International Workshop on Java Technologies for Real-Time and Embedded Systems*, 2011.

[15] Google App Engine. App engine - run your applications on a fully managed paas, 2015. "https://cloud.google.com/appengine" [Accessed March 2015].

[16] Google App Engine Java Sandbox. Google app engine java sandbox, 2015. "https://cloud.google.com/appengine/docs/java/#Java_The_sandbox" [Accessed March 2015].

[17] Google Cloud SDK Service Quotas and Limits, 2015. `https://cloud.google.com/appengine/docs/quotas` [Accessed March 2015].

[18] Google Datastore Fetch Options, 2015. `https://cloud.google.com/appengine/docs/java/javadoc/com/google/appengine/api/datastore/FetchOptions` [Accessed March 2015].

[19] S. Gulwani, S. Jain, and E. Koskinen. Control-flow Refinement and Progress Invariants for Bound Analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[20] S. Gulwani, K. K. Mehra, and T. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2009.

[21] H. He, Z. Ma, H. Chen, and W. Shao. Towards an SLA-Driven Cache Adjustment Approach for Applications on PaaS. In *Asia-Pacific Symposium on Internetware*, 2013.

[22] IEEE APIs. IEEE Xplore Search Gateway, 2015. `http://ieeexplore.ieee.org/gateway/` [Accessed March 2015].

[23] H. Jayathilaka, C. Krintz, and R. Wolski. EAGER: Deployment-time API Governance for Modern PaaS Clouds. In *IC2E Workshop on the Future of PaaS*, 2015.

[24] P. Leitner, B. Wetzstein, F. Rosenberg, A. Michlmayr, S. Dustdar, and F. Leymann. Runtime Prediction of Service Level Agreement Violations for Composite Services. In A. Dan, F. Gittler, and F. Toumani, editors, *Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops*, volume 6275 of *Lecture Notes in Computer Science*, pages 176–186. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-16131-5.

[25] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel. A Fast and Precise Static Loop Analysis Based on Abstract Interpretation, Program Slicing and Polytope Models. In *IEEE/ACM International Symposium on Code Generation and Optimization*, 2009.

[26] K. Mahbub and G. Spanoudakis. Proactive SLA Negotiation for Service Based Systems: Initial Implementation and Evaluation Experience. In *IEEE International Conference on Services Computing*, 2011.

[27] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar. Comprehensive QoS Monitoring of Web Services and Event-based SLA Violation Detection. In *International Workshop on Middleware for Service Oriented Computing*, 2009.

[28] Microsoft Azure Cloud SDK Service Quotas and Limits, 2015. `http://azure.microsoft.com/en-us/documentation/articles/azure-subscription-service-limits/#cloud-service-limits` [Accessed March 2015].

[29] R. Morgan. *Building an Optimizing Compiler*. Digital Press, Newton, MA, USA, 1998. ISBN 1-55558-179-X.

[30] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. ISBN 1-55860-320-4.

[31] D. Nurmi, R. Wolski, and J. Brevik. Model-Based Checkpoint Scheduling for Volatile Resource Environments. In *Proceedings of Cluster 2005*, 2004.

[32] D. Nurmi, J. Brevik, and R. Wolski. Modeling Machine Availability in Enterprise and Wide-area Distributed Computing Environments. In *Proceedings of Europar 2005*, 2005.

[33] D. Nurmi, J. Brevik, and R. Wolski. QBETS: Queue Bounds Estimation from Time Series. In *International Conference on Job Scheduling Strategies for Parallel Processing*, 2008.

[34] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus open-source cloud-computing system. In *IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2009.

[35] ProgrammableWeb. ProgrammableWeb, 2015. `http://www.programmableweb.com` [Accessed March 2015].

[36] F. Raimondi, J. Skene, and W. Emmerich. Efficient Online Monitoring of Web-service SLAs. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2008.

[37] C. Sandberg, A. Ermedahl, J. Gustafsson, and B. Lisper. Faster WCET Flow Analysis by Program Slicing. In *ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems*, 2006.

[38] SearchCloudComputing, 2015. `http://searchcloudcomputing.techtarget.com/feature/Experts-forecast-the-2015-cloud-computing-market` [Accessed March 2015].

[39] J. Skene, D. D. Lamanna, and W. Emmerich. Precise Service Level Agreements. In *International Conference on Software Engineering*, 2004.

[40] K. Stamou, V. Kantere, J.-H. Morin, and M. Georgiou. A SLA Graph Model for Data Services. In *International Workshop on Cloud Data Management*, 2013.

[41] B. Tang and M. Tang. Bayesian Model-Based Prediction of Service Level Agreement Violations for Cloud Services. In *Theoretical Aspects of Software Engineering Conference (TASE)*, 2014.

[42] A. K. Tripathy and M. R. Patra. Modeling and Monitoring SLA for Service Based Systems. In *International Conference on Intelligent Semantic Web-Services and Applications*, 2011.

[43] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A Java Bytecode Optimization Framework. In *CASCON First Decade High Impact Papers*, 2010.

[44] White House APIs. Agency Application Programming Interfaces, 2015. `http://www.whitehouse.gov/digitalgov/apis` [Accessed March 2015].

[45] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-case Execution-time Problem&Mdash;Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7(3), May 2008.

[46] R. Wolski and J. Brevik. QPRED: Using Quantile Predictions to Improve Power Usage for Private Clouds. Technical Report UCSB-CS-2014-06, Computer Science Department of the University of California, Santa Barbara, Santa Barbara, CA 93106, September 2014.

[47] L. Wu, S. Garg, R. Buyya, C. Chen, and S. Versteeg. Automated SLA Negotiation Framework for Cloud Computing. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2013.

[48] E. Yaqub, R. Yahyapour, P. Wieder, C. Kotsokalis, K. Lu, and A. I. Jehangiri. Optimal negotiation of service level agreements for cloud-based services through autonomous agents. In *IEEE International Conference on Services Computing*, 2014.